BurstSketch: Finding Bursts in Data Streams

Ruijie Miao*, Zheng Zhong*, Jiarui Guo*, Zikun Li*, Tong Yang*†, Bin Cui*
* School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
† PCL Research Center of Networks and Communications, Pengcheng Laboratory

***Abstract*—*Burst* is a common pattern in data streams which is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. Burst detection has attracted extensive attention from the research community. To detect bursts accurately in real time, we propose a novel sketch, namely BurstSketch, which consists of two stages. Stage 1 uses the technique Running Track to select potential burst items efficiently. Stage 2 monitors the potential burst items and captures the key features of burst pattern by a technique called Snapshotting. We further propose an optimization, namely Dynamic Buckets, which can improve the accuracy of BurstSketch. We provide theoretical error bounds for Stage 1, Stage 2 and the optimized version. Experimental results show that, compared with the strawman solution, Burstsketch achieves 2.00 to 11.63 times higher F1 score, and 1.56 times higher throughput. We also integrate BurstSketch into Apache Flink, and show that using BurstSketch can be faster than simply using the built-in APIs provided by Apache Flink.**

***Index Terms*—burst, data stream, sketch, approximate query**

I. Introduction

A. *Background and Motivation*

Burst is a common pattern in data streams, which is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. The arrival rate of an item refers to its number of appearances in a fixed time window. Burst is widely regarded as a meaningful structure in diversified fields of data mining. In text data mining, the document streams, such as news articles and research publications, often witness the popularity of a particular topic grow and decline quickly. This results in a burst structure about keywords which are correlated to the topic [2]. By detecting the burst of keywords, we can track the timeline of hot topics, and obtain a general view about the content of the streams. In financial markets, a burst of trading volume may indicate the happening of financial fraud or illegal market manipulation. Further, the burst detection can be applied in clustering [3], [4], online search query analysis [5], [6], web click analysis [2] and social media [7]–[9].

Real-time burst detection in data streams has attracted extensive attention from the research community. In many scenarios, data arrives in a form of data stream. For time-sensitive applications which prefer quick analysis on the data streams, an efficient real-time burst detection algorithm will be more suitable. For example, in electronic commerce systems, real-time burst detection about user queries can provide timely and accurate information about things in vogue or products in demands [10]. Another example is event detection based on social media streaming data. To detect newsworthy events at any given time, an event detection system relies on real-time burst detection algorithms to detect and track burst events in real time. In this paper, we focus on the real-time burst detection in data streams.

The major difficulty of real-time burst detection is to balance the accuracy and the processing speed. The ever-increasing volume of data demands high processing rate. For traditional burst detection algorithms, it is challenging to catch up with high speed(*e.g.*, 20M items per second) of data items while still maintaining high accuracy. To achieve high processing speed, our idea is to maximize cache utilization: ideally, the designed data structure should completely fit in the cache, which can considerably improve the processing speed. To achieve high accuracy of burst detection, our idea is to capture the critical information that are related to bursts. Though the memory consumption of our data structure is limited, the number of bursts is often small compared with the large volume of data items and can be recorded with limited memory. Therefore, our data structure should filter out unrelated information and detect bursts efficiently.

For real-time burst detection in data streams, typical work includes CM-PBE [8] and TopicSketch [9]. CM-PBE can detect bursty events in real time, which is quite efficient in both time and space. TopicSketch aims at detecting hot topics in text streams. It is simple, fast, and easy to deploy. However, these two works define bursts as the sudden increase of frequency, which ignore the sudden decrease. We argue that in some scenarios it is also needed to identify the sudden decrease. For example, prior work [2] has proposed the importance of mining burst structure of keywords that consists of both quick growth and decline. With the detection of frequency decline, it means temporal hot keywords. However, if we only focus on the sudden increase of frequency, it has a quite different meaning: finding keywords that quickly become hot. Simply applying the prior algorithms that are designed to find sudden increase will suffer from low precision in our definition of bursts. In summary, no existing work can provide real-time burst detection in our definition, which consists of a sudden increase and a sudden decrease.

B. *Our Proposed Algorithm*

Towards the design goal of this paper, we propose a novel sketch to accurately detect bursts in real time, namely, Burst-

The first three authors contribute equally.

Tong Yang (yangtongemail@gmail.com) is the corresponding author.

The preliminary version of this paper was published in ACM Special Interest Group on Management of Data (SIGMOD) [1], 2021.

Sketch. To the best of our knowledge, BurstSketch is the first sketch algorithm focusing on detecting bursts in our definition in high speed data streams. BurstSketch has the following features.

- BurstSketch is memory efficient: it is small enough to be held in CPU L2 caches.
- BurstSketch is accurate: it achieves higher than 97% F1 score (using 60 KB memory), which is 2.00 to 11.63 times higher than the strawman solution.
- BurstSketch is fast: the time complexity for insertion and query is $O(1)$, and it achieves throughput higher than 20 Mips.

we design two versions of BurstSketch: the basic version and the optimized version. The basic version of BurstSketch consists of two parts, Stage 1 and Stage 2. For each incoming item, we first check whether it is a potential burst item in Stage 1, if so, it will be sent to Stage 2. The techniques used in Stage 1 and Stage 2 are named Running Track and Snapshotting, respectively. We show the two key techniques below.

Technique I: Running Track. Running Track is used to select potential burst items. It needs to filter out infrequent items as well as items arrive at a steady speed. Running Track works as follows. We use many tracks, each item will be mapped into d tracks by hash functions $h_1(.) \dots h_d(.)$. For each track, we only observe the most frequent item. If it is frequent enough, we consider it as a potential burst item. To find the fastest item in each track, there are several optional strategies: *frequent* [11], *probabilistic decay* [12], *probabilistic replacement* [13]. We choose *frequent* since it is the simplest and fastest which has a comparative accuracy compared to others. In our strategy, high-speed items are unlikely to be filtered out in every track, because it would be selected as long as it becomes the most frequent item in at least one track.

Technique II: Snapshotting. Snapshotting is used to detect bursts from potential bursts. The rationale of Snapshotting is that a burst can be described only with the sudden increase and sudden decrease in arrival rate. Therefore, we do not need to record frequencies of items in every time window. In Snapshotting, we only take two snapshots for the sudden increase and the sudden decrease so that we can confirm whether it is a burst. Snapshotting detects bursts with $O(1)$ memory.

We propose the optimized version based on the observation of the high skewness in real-world datasets: most items are infrequent, and a small amount of frequent items make up the majority of the total frequency. Prior work [14] indicates that such high skewness is common in the real-world scenarios. For most infrequent items, the Stage 1 in the basic version only records in the counters a small count. Even for frequent items, as they will be regarded as potential burst items and reported to Stage 2, they can be recorded with a small counter with high probability. Therefore, we utilize the novel idea of Dynamic Bucket to improve the memory efficiency. Dynamic Bucket uses fingerprint to substitute ID, and supports dynamic adjustment of memory partition for the fingerprint and the counter. At the beginning, the bucket uses a long fingerprint and a small counter. As the value in the counter grows, the bucket will transition to use a short fingerprint and a large counter. The optimized version incorporates more complex operations, sacrificing a little processing speed for improvement in accuracy. When the memory consumption is 20KB, the throughput of the optimized version drops from 27MIPS to 24MIPS, while the recall improves from 87% to 93%. As the optimized version still maintains a high processing speed, it is suitable for the scenarios where operators have a stricter demand of accuracy for burst detection.

The BurstSketch can be applied in the distributed scenarios by deploying in the Apache Flink. Detecting bursts by simply using the built-in APIs of Flink requires recording and processing frequency information for all IDs. In contrast, BurstSketch consumes limited memory in each worker and is more cache friendly. As a result, BurstSketch can achieve higher throughput when applied in the Flink.

C. Main Contributions

- We propose BurstSketch and the optimized variants for burst detection, which are accurate, efficient, and consumes limited memory.
- We theoretically analyze the basic version and the optimized version of BurstSketch, and prove that each stage can find burst with bounded error.
- We conduct rich experiments on a variety of datasets. The results of experiments demonstrate the high performance of BurstSketch, and the improvement of its optimization.
- We explore the possibility of deploying BurstSketch to Apache Flink and show that BurstSketch can be applied in distributed scenarios.

II. Problem Statement & Related Work

A. Problem Statement

The symbols frequently used in this paper are shown in Table I.

TABLE I: Symbols Used in This Paper

Notation	Meaning
$\mathcal{A}_i$	i^{th} bucket array of Stage 1
$\mathcal{B}$	Bucket array of Stage 2
k	parameter for the definition of sudden increase and sudden decrease
L	Maximum width of a burst
T	Burst threshold
H	Running Track threshold
C_{pre}	Frequency in the previous time window
C_{cur}	Frequency in the current time window
t	Timestamp in Stage 2

Burst Detection: *Burst*, in our definition, is a particular pattern of the changing behavior in terms of the arrival rate of an item in a data stream, and the pattern consists of a sudden increase and a sudden decrease. Specifically, we divide the data stream into fixed-width time windows. Given an item e, a sudden increase means that, in two adjacent time windows, the arrival rate of e in the second time window is no less than k times of that in the first time window. Similarly, a sudden decrease is that the arrival rate of e in the second time window is no more than $\frac{1}{k}$ of that in the first time window. Also, we do not consider infrequent bursty items as bursts, for they are not useful in most applications, so the arrival rate of a burst item should exceed a *burst threshold*. In practice, a burst occurs over a short period of time. Therefore, we set a limitation L for the width of a burst, namely, the number of time windows that the burst lasts. The formal definition of a burst is as follows. **Formal Definition:** For a time series data stream $S = \{e_{t1},\ e_{t2},\ e_{t3},\ \ldots\}$, an item e and a burst threshold T, given that the data stream is divided into fixed time windows $w_1,\ w_2,\ w_3,\ \ldots$ and the arrival rate of e in the time windows are $r_1,\ r_2,\ r_3,\ \ldots$, if there exist four time windows $w_i,\ w_{i+1},\ w_j,\ w_{j+1}$, where

$$r_{i+1} \geqslant k \cdot r_i \wedge r_{j+1} \leqslant \frac{1}{k} \cdot r_j \wedge j > i$$

and

$$r_k \geqslant T, \forall k \in \{i+1,\ \ldots,\ j\} \wedge j - i \leqslant L$$

then e is a burst item, the changing process of its arrival rate is a burst, the width of the burst is $j - i$ time windows, window w_{i+1} is the sudden-increase window, and window w_{j+1} is the sudden-decrease window. If multiple sudden-increase windows happen consecutively, we just consider the latest one as the burst's possible beginning. If multiple sudden-decrease windows happen consecutively, we just consider the first one after the sudden increase as the burst's end. It can prevent multiple reports of a single burst.

B. The Comparison of the Definitions of Burst

Kleinberg's work [2] models the burst structure using an infinite-state automation, where each state denotes a level of message emitting rate. If the automation starts from the low-rate states, transits to high-rate states, and finally falls back to low-rate states, a burst is detected. The definition in Kleinberg's work is similar to our definition, as we all define the burst in a rising and declining manner. Kleinberg's definition does not restrict the rising and declining rate and hence is not designed for sudden increase and sudden decrease, while our definition restricts the rate by the parameter k. We argue that it is hard to pre-define the exact range of high rates and low rates, so we use parameter k to describe the amplitude of the increase and decrease.

Some work [15] defines burst as a large number of events occurring in a certain time window. Some other work [8], [9] defines the burst as the sudden increase in frequency, and does not care about whether there is a sudden decrease. Our definition in this paper is more complete with both sudden increase and sudden decrease. We believe all above definitions refer to different data patterns, and thus have different applications. In some applications, the occurrence of decreasing trend is also crucial. Kleinberg's work [2] has pointed out the importance of detecting both increase and decrease for nested burst structure in text mining. In electronic commerce systems, the decreasing trend should be considered. If the sales of an item contains both a sudden increase and a sudden decrease, the item should be regarded as temporally hot, which should be considered in the business decisions. Another example is network management. The flows that consist of both sudden rate increase and sudden rate decrease are different from those with only sudden increase. For the flows that match our definition of bursts, they consume resources in a short period of time. The operators need to identify burst flows and their period in order to allocate the required resources for burst flows.

C. Prior Work on Burst Detection

Abundant algorithms on burst detection [2], [8], [9], [16]–[18] have been proposed. We will list those with close relation to our work in the follows. Besides, research work [19]–[24] on finding frequent items has close relation to burst detection, and will be briefly discussed.

Kleinberg [2] models bursts as state transitions, and utilizes Bayes procedure to compute the optimal state sequence. The state sequence will then be converted to tree representation and extracted bursty structure. Their definition of burst, as mentioned in II-B, is similar to ours. Nonetheless, it is a non real-time algorithm. And in their algorithm, each automation is correlated with exactly one item, which means that the space consumption grows linearly with the number of items under monitoring. In contrary, BurstSketch is a real-time algorithm with limited memory consumption.

CM-PBE [8] is a recent work concerning burst detection, which concentrates on detecting burst from history without storing or querying the whole stream. To identify bursty events in data streams, they propose a concept named *frequency curve*, which shows how the item's frequency grows cumulatively over time. They propose two algorithms: CM-PBE-1 and CM-PBE-2. To approximate the curve, CM-PBE-1 uses dynamic programming, while CM-PBE-2 solves linear programming. Both algorithms can largely save the storage space, as they store as few points as possible. Our algorithm differs in two regards. First, the definitions of bursts are different. In their work, an event that witnesses a large acceleration in its arrival rate is considered a bursty event, whereas in our definition, burst consists of a sudden increase and a sudden decrease in its arrival rate. Besides, our algorithm cares about real-time burst detection in high speed data streams, while their work puts a premium on bursty events detection in history.

TopicSketch [9] focus on detecting burst in real time. They also use the acceleration of items' arrival rate as a metrics of burst. To calculate the acceleration, they incrementally maintain velocities of two time windows. Their definition of burst is close to the definition of CM-PBE, which is different from ours, as mentioned above.

Research work on finding frequent items in unbounded data streams is also closely related, as our definition requires the item frequency in the burst period higher than the burst threshold T. Tong et. al [20] study two different definitions under the scenarios of uncertain data: expected support-based frequent itemset and probabilistic frequency itemset. They clarify the relationship between these two definitions, provide uniform baseline and evaluate all existing representative algorithms. Jin et. al [19] propose a novel framework for sliding window top-k queries on uncertain streams. They carefully design the synopsis that achieves the same asymptotic processing time bound as the base synopsis, but much lower asymptotic space bound. These two work focuses on finding frequent items in uncertain data streams. For burst detection in the uncertain data streams, the methods in the above literature can be used to track frequent items in the time window, and then further analysis on burst detection can be applied. This paper focuses on the scenarios of certain data streams, and burst detection in the uncertain data streams is left for future work.

D. Typical Sketch Algorithms

In order to detect burst efficiently, our proposed algorithm takes advantage of the technique of sketches. Sketches are probabilistic algorithms for data stream processing. They can generate approximate answers to queries with small memory consumption and very high speed. Due to the above advantages, sketches are applied to a wide range of data stream processing tasks, such as finding frequent items [25]–[30], finding top-k hot items [11]–[14], [31]–[33], detecting heavy changes [34]–[36], graph stream summarizing [37], [38], and item classification [39]. Moreover, sketches are also applied in many other areas [40]–[48], such as machine learning [49], membership testing [50], persistent data structures [51].

III. The BurstSketch Algorithm

In this section, we propose the BurstSketch algorithm. First, we introduce the strawman solution in Section III-A. Second, we introduce the BurstSketch algorithm in Section III-B. Finally, we introduce the optimization on BurstSketch in Section III-C.

A. The Strawman Solution

The strawman solution is based on the CM sketch. The CM sketch consists of k counter arrays, each associated with a hash function. For each incoming item, the hash function is calculated to map it to a mapping bucket in each array, then all the mapping buckets of the item is increased by 1. To report the estimated frequency of an item, the CM sketches output the minimum value among the mapping buckets. In the strawman solution , we construct $L+2$ CM sketches to store the estimated frequencies of the latest $L+2$ time windows to detect burst whose width no larger than L. We use a queue to store potential burst items. Whenever the frequency of an item in a window is larger than the burst threshold, we insert its flow ID into the queue. At the end of each time window, for potential burst items, we query their frequencies from CM sketches to find burst patterns. Although the strawman solution is capable to detect bursts, it is memory consuming and inaccurate. Because it stores information of $L+2$ windows and takes into account many items that are not potential bursts.

B. The BurstSketch Algorithm

Rationale: In this paper, we propose a novel sketch, namely BurstSketch. BurstSketch consists of two stages. To avoid recording unnecessary information, the first stage checks whether an incoming item is a potential burst item. We only send the potential items to the second stage for burst detection. To detect a burst, rather than recording the frequencies of $L+2$ time windows for each item, Stage 2 only records the frequencies of 2 adjacent time windows for potential burst items to detect whether there exists sudden increase or sudden decrease, and we use a timestamp to snapshot it. In summary, compared to the strawman solution, our BurstSketch filters out much more unnecessary information.

Data Structure: As shown in Figure 1, BurstSketch has two stages: Stage 1 using **Running Track** to filter low arrival rate items, and Stage 2 using **Snapshotting** to find burst patterns. Stage 1 consists of d bucket arrays $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_d$, and each array consists of m buckets. There are d hash functions $h_1(.), h_2(.), \ldots, h_d(.)$ associating with d bucket arrays respectively. Each bucket has two fields: item ID (key) and frequency. We have a Running Track threshold H to determine whether the item is a potential burst item. It is worth noting that the number of tracks determines the maximum number of bursts our BurstSketch can detect simultaneously. A single track takes up only several bytes, but more tracks enable us to detect more bursts simultaneously, and also lessens hash collisions. Therefore, we recommend using enough tracks to achieve higher accuracy. Stage 2 is a bucket array $\mathcal{B}[1], \mathcal{B}[2], \ldots, \mathcal{B}[M]$ associated with a hash function $g(.)$. Each bucket has s cells. Each cell has four fields: item ID (key), two counters C_{pre} and C_{cur}, timestamp t. C_{pre} is used to record the frequency of the item in the previous time window, while C_{cur} is used to record the frequency of the item in the current time window. The timestamp records the time window in which the latest sudden increase happened. If the timestamp is equal to 0, it means no sudden increase occurred.

Insertion: Given an incoming item e, if e is in Stage 2, we increment $e.C_{cur}$ by 1. Otherwise, we insert it into Stage 1: we hash e into d mapping buckets of Stage 1 $\mathcal{A}_1[h_1(e)], \mathcal{A}_2[h_2(e)], \ldots, \mathcal{A}_d[h_d(e)]$. For each bucket, there are 3 cases.

Case 1: e is not in the bucket, and the bucket is empty. In this case, we insert e into the bucket with the frequency of 1.

Case 2: e is not in the bucket, and the bucket is not empty. In this case, we need a replacement strategy to allow a new potential burst to get in. We apply Frequent [11]: we decrement the frequency of the bucket by 1. If the frequency is decreased to 0, we empty the bucket. There are two other typical replacement strategies, namely, probabilistic decay [12], and probabilistic replacement [13]. For probabilistic decay, suppose the recorded frequency is f, we decrement the frequency by 1 in the probability of $f^{-1.08}$. If the frequency is decrease to 0, we empty the bucket. For probabilistic replacement, with a probability of $\frac{1}{f+1}$, we replace the recorded ID by e and increase the counter by 1. We choose Frequent because it is fast and easy to implement, and it can efficiently evict infrequent items and save buckets for burst items.

Case 3: e is in the bucket. We just increment the frequency of e by 1. If the frequency of e is equal to or larger than the Running Track threshold H, we try inserting e into Stage 2 (because e is frequent enough): if we find an empty cell in the bucket $\mathcal{B}[g(e)]$, we insert e in it with its frequency. Otherwise, we try evicting the smallest item whose timestamp t is 0: if the frequency of the item is smaller than the frequency of e

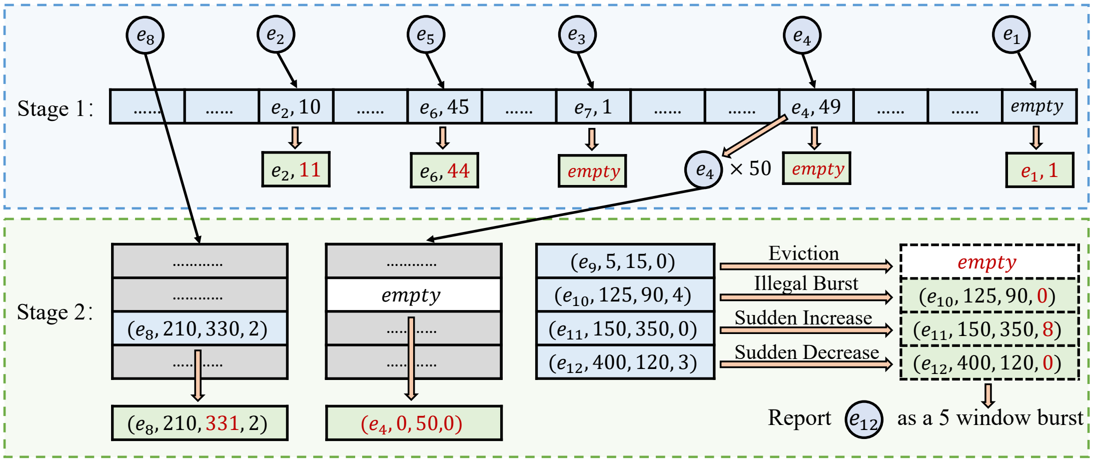


Fig. 1: An example of BurstSketch using one hash function.

Algorithm 1: Insertion-BurstSketch

```
Input: an item $e$; $H$, the Running Track threshold;
if $e$ is in $\mathcal{B}[g(e)]$ then
    $e.C_{cur} \leftarrow e.C_{cur} + 1$;
else
    for each $i \in [1, d]$ do
        if $e$ is in $\mathcal{A}_i[h_i(e)]$ then
            increase the frequency of $e$ by 1;
            if the frequency of $e \geq H$ then
                if Insert_Stage2 ($e$, the frequency of $e$)
                  then
                    clear $\mathcal{A}_i[h_i(e)]$ to empty;
        else if $\mathcal{A}_i[h_i(e)]$ is empty then
            insert $e$ into $\mathcal{A}_i[h_i(e)]$ and set the
              frequency of $e$ to 1;
        else if $e$ is not in $\mathcal{A}_i[h_i(e)]$ and $\mathcal{A}_i[h_i(e)]$ is
          not empty then
            decrease the frequency of $\mathcal{A}_i[h_i(e)]$ by 1;
            if the frequency of $\mathcal{A}_i[h_i(e)]$ is 0 then
                clear $\mathcal{A}_i[h_i(e)]$ to empty;
Function Insert_Stage2($e$, $C$):
    if $C >$ the $C_{cur}$ of the smallest item in $\mathcal{B}[g(e)]$
      then
        use $e$ to replace the smallest item;
        $e.C_{cur} \leftarrow C$; $e.C_{pre} \leftarrow 0$;
        return 1;
    return 0;
```

, we evict the item and insert e with its frequency. If all the items' t are not 0, we try evicting the smallest item in the bucket with the same method. Stage 2 stores and monitors potential bursts. The space in Stage 2 is limited, so we need to evict the items that are not likely to become a burst when the corresponding bucket is full.

Detection: Stage 2 uses Snapshotting to capture the sudden increase and the sudden decrease for each item, and reports bursts in the end of each time window. For item e, suppose the max width of a burst is L. First we detect if there is a sudden increase or sudden decrease: we check the frequencies of e in the latest two time windows. If $\frac{e.C_{cur}}{e.C_{pre}} \geq 2$, the sudden increase happens. Then we update the current time window into t. Specially, if e has been inserted into Stage 2 in the current time window (which means we do not know $e.C_{pre}$), we regard $e.C_{pre}$ as 0. If $\frac{e.C_{cur}}{e.C_{pre}} \leq \frac{1}{2}$, a sudden decrease happens. Then we check whether there has been a sudden increase and whether the difference between t and the current time window is no more than L. If so, BurstSketch reports a burst which has t as its sudden-increase window and the current time window as its sudden-decrease window, then we clean $e.t$ to 0. Otherwise, no burst is reported and t remains unchanged.

Cleaning Policy: In Stage 1, we clean all arrays at the end of each time window. In Stage 2, we evict the items whose arrival rates are always low. Specifically, at the end of every time window, we check if the frequencies of the latest two time windows are both lower than H. If so, we evict the item. We also clean illegal potential bursts, whose frequency is smaller than T in the current time window. If so, we clean t of the item to 0.

A Running Example: Figure 1 shows a running example of BurstSketch. In this example, in Stage 2, given a bucket with $(e_{10}, 125, 90, 4)$, e_{10} is the item ID, 125 is e_{10}'s frequency in the previous time window C_{pre}, 90 is e_{10}'s frequency in the current time window C_{cur}, and 4 is the time when the latest sudden increase happens. Suppose the Running Track threshold $H = 50$, the burst threshold $T = 100$, and the time of this example is at the end of time window 8.

Example 1: To insert e_8, we find it in Stage 2, so we just increment $e_8.C_{cur}$ by 1.

Example 2: To insert e_2, we find it in Stage 1, so we just increment the frequency of e_2 by 1.

Example 3: To insert e_5, we do not find it in both stages, so we decrement the frequency of the item in the mapped bucket by 1.

Example 4: To insert e_3, we decrement the frequency of e_7 from 1 to 0, then we evict e_7.

Example 5: To insert e_4, we find it in Stage 1, so we increment e_4 by 1. After the increment, the frequency of e_4 reaches the Running Track threshold and we find an empty cell in Stage 2. Then we clean e_4 in Stage 1 and insert it into Stage 2 with the frequency of 50.

Example 6: To insert e_1, we find an empty bucket in Stage 1, so we insert e_1 with the frequency of 1.

At the end of every time window, we check if there is any sudden increase, sudden decrease, illegal burst, or legal burst. At the same time, we evict the items which are not potential burst items anymore.

Example 7: For e_9, both $e_9.C_{pre}$ and $e_9.C_{cur}$ are below 50, so we evict e_9 from Stage 2.

Example 8: For e_{10}, $e_{10}.C_{cur}$ is below 100, it means it is an illegal burst, so we clean its timestamp to 0.

Example 9: For e_{11}, $\frac{e_{11}.C_{cur}}{e_{11}.C_{pre}} = \frac{350}{150} \geq 2$, it means a sudden increase happens. Therefore, we record the current time window 8 into the timestamp field.

Example 10: For e_{12}, $\frac{e_{12}.C_{cur}}{e_{12}.C_{pre}} = \frac{120}{400} \leq \frac{1}{2}$, it means a sudden decrease happens. And we find the width of the burst (*i.e.*, $8 - 3 = 5$) is legal. Therefore, we report e_{12} as a burst with a width of 5. Then we clean the timestamp of e_{12}.

Bursts inside bursts: We have an extended version to detect bursts inside bursts. The definition of bursts inside bursts is similar to *bracket matching*: sudden increase corresponds to left bracket and sudden decrease corresponds to right bracket. To detect bursts inside bursts, the ideal algorithm works as follows. We add a stack for each item in Stage 2. When a sudden increase happens, we push a timestamp with current time into the stack. If the stack is full, we delete the oldest timestamp, which is at the bottom of the stack. We use an array with two pointers (a header and a tail) to implement the stack, and thus can delete the timestamp from the bottom of the stack. When a sudden decrease happens, we pop the timestamp (the most recent sudden increase) from the top of the stack, and report the pair of sudden increase and sudden decrease as bursts inside bursts. If the stack is empty, we do nothing.

C. Optimization: Dynamic Buckets

Rational: Stage 1 in BurstSketch is used to filter out infrequent items and report potential burst items to Stage 2. We notice that, for infrequent items, as the frequencies are low, they do not need large counters. For potential burst items, as the algorithm tries to report them to Stage 2 when the frequencies reach Running Track threshold H, they also do not need large counters with high probability. However, relatively larger counters are needed compared with infrequent items. To achieve memory efficiency, we can use smaller counters in the buckets. Moreover, we wish the infrequent items can be stored in small counters, while the burst items can be stored in relatively larger counters.

Therefore, we propose an optimization named *Dynamic Bucket*, which is designed to optimize Stage 1. In the optimization, instead of recording full IDs in the original buckets in Stage 1, we record fingerprint, which is the hash value of the full item ID and has less bit width. We also use less bits as counters, so the bucket size is much smaller. Our main idea is to divide the fingerprint field and counter field dynamically, in order to allocate different size of counters for different items. At the beginning of each time window, each bucket will use a division strategy that allocate more bits to record fingerprint, and less bits to counters. As the items are inserted, the frequency in the bucket grows. Once the counter is going to overflow, the bucket is re-divided, and we use more bits as counter and less bits as fingerprint. In this way, we allocate larger counters for burst items and smaller counters for infrequent items.

Data Structure: For the optimized version, the Stage 2 is the same as the original BurstSketch. Therefore, we only describe the data structure of Stage 1. In the optimization, the Stage 1 contains d arrays $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_d$, and each array consists of m *dynamic buckets*. As shown in figure 2, a dynamic bucket is composed of a fingerprint field, a counter field, and an indicator field. The sum of the fingerprint field's width and the counter field's width is a fixed number w, and several division schemes are predefined to decide how w bits are allocated to both fields. The information about which division schemes the dynamic bucket uses currently is encoded in the indicator field. Suppose there are D predefined division schemes, then $\lceil log(D) \rceil$ bits is needed for the indicator field in each dynamic bucket. We sort the division schemes according bit width of the counter field. The i-th scheme allocate c_i bits for counter field and p_i bits for fingerprint field, and we encode it as i in the indicator field. Same as the original BurstSketch, there is d hash functions $h_1(.), h_2(.), \ldots, h_d(.)$ associating with d arrays respectively. In addition, another hash function $h_{fp}(.)$ is needed to calculate the fingerprint for item IDs, which takes IDs as input and output fingerprints of width w.

Insertion: Given the incoming item e, if e is in Stage 2, we insert it to Stage 2. Otherwise, we insert e into Stage 1. We compute the fingerprint $h_{fp}(e)$ of the item e, and hash it into d mapping buckets of Stage 1, $\mathcal{A}_1[h_1(e)], \mathcal{A}_2[h_2(e)], \ldots, \mathcal{A}_d[h_d(e)]$.

For each hashed bucket, suppose the current indicator field is i. There are 3 cases for insertion.

Case 1: the bucket is empty. In this case, we truncate $h_{fp}(e)$ and use lower p_i bits to set the fingerprint field, and then set the counter field to 1.

Case 2: the bucket is not empty, and lower p_i bits of $h_{fp}(e)$ do not match with the fingerprint field. In this case, the counter field is decreased by 1. If the counter field become 0 after the decrease, we empty the bucket.

Case 3: the bucket is not empty, and lower p_i bits of $h_{fp}(e)$ match the fingerprint field. In this case, we should increase the counter field by 1. However, if the counter field is going to overflow, we should move to the $i+1$ division scheme before increasing the counter: re-divide the dynamic bucket, allocate c_{i+1} bits for the counter field and f_{i+1} bits for the fingerprint field, and set the fingerprint field to lower f_{i+1} bits of $h_{fp}(e)$, set the counter field to the original value. When the frequency is equal or larger than the Running Track threshold H, we try to insert e into Stage 2, and clear it in Stage 1 if succeeded.

Cleaning Policy: For each dynamic bucket in Stage 1, we set both the counter field and fingerprint field to 0. Besides, we

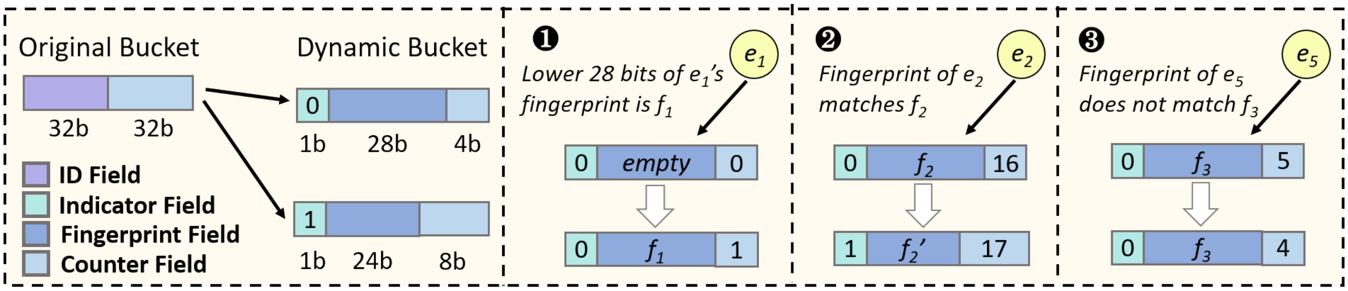


Fig. 2: Dynamic Buckets in the Stage 1 of the optimized BurstSketch.

set the indicator to 0, as the corresponding division scheme has the smallest counter field.

A Running Example: Figure 2 shows a running example of BurstSketch with the optimization. In the example, the sum w of the fingerprint field's width and the counter field's width is 32. Two division schemes are predefined: if the indicator field is 0, it indicates that the fingerprint field is 28 bits and the counter field is 4 bits; if the indicator field is 1, it indicates that the fingerprint field is 24 bits and the counter field is 8 bits. Compared with the original bucket in Stage 1, the dynamic bucket is much smaller. We use the same division schemes in our evaluations, because we set Running Track threshold H to 50 and an 8-bit counter is enough.

Example 1: To insert e_1, we compute the fingerprint $h_{fp}(e_1)$. We find that the indicator is 0, and the bucket is empty, so we set the fingerprint field to lower 28 bits of $h_{fp}(e_1)$ (f_1) and set the counter field to 1.

Example 2: To insert e_2, we compute the fingerprint $h_{fp}(e_2)$. We find that the indicator is 0, and the lower 28 bits of $h_{fp}(e_2)$ match f_2, so we want to increase the counter field by 1. However, the counter field is going to overflow. So we re-divide the two fields, set the indicator to 1, and increase the counter field to 17.

Example 3: To insert e_5, we compute the fingerprint $h_{fp}(e_5)$. We find that the lower 28 bits of $h_{fp}(e_5)$ do not match f_3, so we decrease the counter field to 4.

IV. Mathematical Analysis

In this section, we provide theoretical analysis for BurstSketch. First, we derive the error bound of Stage 1 in Section IV-A. Then we show an upper bound of the number of distinct items in Stage 2 in Section IV-C. Finally, we show that there is no overestimation error in Section IV-D.

A. The Error Bound of Stage 1

Lemma 1. *Given a time series data stream S which has fixed window size. In a window w, for item e_i, suppose e_i does not in Stage 2. Let $F_{i,j,k}$ be the number of items mapping to bucket $\mathcal{A}_j[k]$ in w except for item e_i, f_i be the frequency of e_i in w, $\mathcal{A}_j[k].ID$ be the ID of bucket $\mathcal{A}_j[k]$, $\mathcal{A}_j[k].count$ be the frequency of bucket $\mathcal{A}_j[k]$. Suppose $f_i > F_{i,j,k}$, which means e_i is in the majority in this bucket, we have $\mathcal{A}_j[k].ID = e_i$ and $f_i - F_{i,j,k} \leqslant \mathcal{A}_j[k].count \leqslant f_i$.*

Proof. Since each item which is not e_i can at most counteract one e_i, so there at least remains $f_i - F_{i,j,k}$ numbers of e_i. Therefore, $\mathcal{A}_j[k].ID = e_i$ and $f_i - F_{i,j,k} \leqslant \mathcal{A}_j[k].count$. $\mathcal{A}_j[k].count \leqslant f_i$ is obvious because $\mathcal{A}_j[k].count$ increases only when the item is equal to $\mathcal{A}_j[k].ID$. □

Theorem 2. *Given a time series data stream S which has fixed window size W. In a window w, suppose $\mathcal{A}_j[k].ID = e_i$, let f_i be the frequency of item e_i in w. For $0 < \varepsilon < f_i$, we have*

$$Pr\{f_i - \mathcal{A}_j[k].count \geq \varepsilon\} \leqslant \frac{W - f_i}{m\varepsilon} \tag{1}$$

Proof. By the linearity of the expectation and the pairwise independence of the hash functions, we have

$$E[F_{i,j,k}] = E[\sum_{e \neq e_i} f_e I_{h_j(e)=h_j(e_i)}] = \sum_{e \neq e_i} f_e \frac{1}{m} = \frac{W - f_i}{m}$$

where f_e is the frequency of item e in the window. By Markov inequality, we have

$$Pr\{F_{i,j,k} < \varepsilon\} = 1 - Pr\{F_{i,j,k} \geq \varepsilon\} \geq 1 - \frac{W - f_i}{m\varepsilon}$$

Therefore, according to the lemma above,

$$\begin{aligned} Pr\{f_i - \mathcal{A}_j[k].count \geq \varepsilon\} &= 1 - Pr\{f_i - \mathcal{A}_j[k].count < \varepsilon\} \\ &\leqslant 1 - Pr\{f_i > F_{i,j,k} \wedge F_{i,j,k} < \varepsilon\} \\ &= 1 - Pr\{F_{i,j,k} < \varepsilon\} \\ &\leqslant \frac{W - f_i}{m\varepsilon} \end{aligned}$$

□

B. The Error Bound of Optimized Stage 1

Theorem 3. *Given a time series data stream S with fixed window size W. Suppose D division schemes are predefined, and i-th scheme allocate c_i bits for counter field and p_i bits for fingerprint field. In a window w, suppose $h_{fp}(e_i)$ matches $\mathcal{A}_j[k].fingerprint$, let f_i be the frequency of item e_i in w. For $\varepsilon > 0$, we have*

$$Pr\{|f_i - \mathcal{A}_j[k].count| \geq \varepsilon\} \leqslant \frac{W - f_i}{m2^{p_D}\varepsilon} + \frac{(W - f_i)(2^{p_0} - 1)}{m2^{p_0}\varepsilon} \tag{2}$$

Proof. Let $F_{i,j,k}$ be the number of items that maps to bucket $\mathcal{A}_j[k]$ and match the fingerprint field other than e_i. Let $G_{i,j,k}$ be the number of items that maps to bucket $\mathcal{A}_j[k]$ and do not match the fingerprint field other than e_i. By the linearity of the expectation of the pairwise independence of the hash function, we have

$$\begin{aligned} E[F_{i,j,k}] &= E[\sum_{e\neq e_i} f_e I_{h_j(e)=h_j(e_i)\wedge h_{fp}(e)=h_{fp}(e_i)}] \\ &\leqslant \sum_{e\neq e_i} f_e \frac{1}{m2^{p_D}} = \frac{W-f_i}{m2^{p_D}} \end{aligned}$$

$$\begin{aligned} E[G_{i,j,k}] &= E[\sum_{e\neq e_i} f_e I_{h_j(e)=h_j(e_i)\wedge h_{fp}(e)\neq h_{fp}(e_i)}] \\ &\leqslant \sum_{e\neq e_i} f_e \frac{1}{m} \cdot (1-\frac{1}{2^{p_0}}) = \frac{(W-f_i)(2^{p_0}-1)}{m2^{p_0}} \end{aligned}$$

where p_0 is the largest fingerprint field, and p_D is the smallest fingerprint field. By Markov inequality, we have

$$\begin{aligned} Pr\{f_i - \mathcal{A}_j[k].count \geq \varepsilon\} &\leqslant \frac{E[f_i - \mathcal{A}_j[k].count]}{\varepsilon} \\ &\leqslant \frac{E[F_{i,j,k}]}{\varepsilon} \\ &\leqslant \frac{W-f_i}{m2^{p_D}\varepsilon} \end{aligned}$$

$$\begin{aligned} Pr\{\mathcal{A}_j[k].count - f_i \geq \varepsilon\} &\leqslant \frac{E[\mathcal{A}_j[k].count - f_i]}{\varepsilon} \\ &\leqslant \frac{E[G_{i,j,k}]}{\varepsilon} \\ &\leqslant \frac{(W-f_i)(2^{p_0}-1)}{m2^{p_0}\varepsilon} \end{aligned}$$

Therefore, we have

$$\begin{aligned} &Pr\{|f_i - \mathcal{A}_j[k].count| \geq \varepsilon\} \\ \leqslant &Pr\{f_i - \mathcal{A}_j[k].count \geq \varepsilon\} + Pr\{\mathcal{A}_j[k].count - f_i \geq \varepsilon\} \\ \leqslant &\frac{W-f_i}{m2^{p_D}\varepsilon} + \frac{(W-f_i)(2^{p_0}-1)}{m2^{p_0}\varepsilon} \end{aligned}$$

□

C. Upper Bound of the Number of Distinct Items in Stage 2

Theorem 4. *Given a data stream S. We assume each window has W items. In each window, S obeys an arbitrary distribution. Let n be the number of distinct items in Stage 2, H be the Running Track threshold. Then, we have*

$$n \leqslant \frac{3W}{H} \tag{3}$$

Proof. For an item, it is in Stage 2 either because it has already been in Stage 2 before this window or because it passes through Stage 1 in this window. We denote f_0 the frequency of the item in the current window, f_1 the frequency of the item in the previous window, f_2 the frequency of the item in the window before the previous window. In the case of the item that has already been in Stage 2, because of the cleaning policy, we have $f_1 \geq H \vee f_2 \geq H$. In another case, the item passes through Stage 1, which means $f_0 \geq H$. In summary, for an item in Stage 2, it satisfies $f_0 \geq H \vee f_1 \geq H \vee f_2 \geq H$. For each window, the number of items whose frequency is not less than the threshold is no more than $\frac{W}{H}$. We add up it and derive the upper bound $\frac{3W}{H}$. □

D. Proof of no Overestimation Error

Theorem 5. *For any item e_i in Stage 2, let $\hat{f_i}$ be the estimated frequency of item e_i in Stage 2, f_i be the real frequency, then*

$$\hat{f_i} \leqslant f_i$$

Proof. For item e_i, if it has already been in Stage 2 before the current window, it is obvious that estimated frequency $\hat{f_i}$ is equal to the real frequency f_i. If it passes through Stage 1 in the current window, the frequency before being stored in Stage 2 should not be less than the Running Track threshold. Because we set the threshold as the initial value of $\hat{f_i}$, we have $\hat{f_i} \leqslant f_i$. □

V. Deployment of BurstSketch on Apache Flink

Apache Flink [52] is one of the state-of-the-art data stream processing frameworks. A popular task on Flink is to conduct real-time data stream analytics with low latency, and integrating sketches into Flink has received attention in recent researches. For example, Condor [53] implements CM sketches [25], HyperLogLog [54], DDSketch [55], *etc*. Based on the sketch implementation, Condor supports the processing of synopsis-based streaming jobs on the top of Flink. Some other works [56]–[58] also propose the implementation of sketch-based solutions on the top of Flink and evaluate their performance. The motivation for deploying sketches on Flink is as follows. First, sketch-based solutions can accelerate data processing while providing guaranteed accuracy, and thus is suitable for many tasks. Second, by deploying on Flink and utilizing the APIs provided by Flink, the sketched-based solutions can work in distributed scenarios.

In this section, we first describe a naïve solution to detect bursts that match our definition with the built-in method in Flink, and then we describe the implementation of BurstSketch on Flink.

A Naïve Solution: Flink provides APIs for state management while processing data streams. To detect burst corresponding to our definition, a naïve solution is to maintain stateful information for each key, which includes the frequency in current window, the frequency in last window, *etc*. We call this solution *Stateful Detector*. For implementation, we extend *KeyedProcessFunction*, and use a *ValueState* to record all stateful information.

BurstSketch: For implementation of BurstSketch on Flink, we extend *KeyedProcessFunction* and maintain BurstSketch in the *KeyedProcessFunction*. As items with the same key are sent to the same instance of *KeyedProcessFunction*, they will be inserted into the same BurstSketch that is owned by the instance. This guarantees the correctness of the implementation.

VI. Experimental Results

In this section, we show the experimental results of BurstSketch. First, we describe the experimental setup in Section VI-A. Second, we show how parameter settings affect the performance of BurstSketch and the optimized version in Section VI-B and Section VI-C, respectively. Third, in Section VI-D, we evaluate the performance of BurstSketch and the optimized version in three datasets, and compare them with the strawman solution and prior works. Then, we provide analyses on BurstSketch and the optimized version in Section VI-E. Finally, we conduct experiments on Apache Flink.

A. Experimental Setup

Datasets: We use the following datasets in our experiments and divide them into count-based windows and time-based windows.

1) IP Trace Dataset: As many papers [12], [31] do, we use anonymized IP trace streams from CAIDA [59]. CAIDA identifies each flow of IP trace streams by the five-tuples: source and destination IP address, source, and destination port, protocol. We use the source and destination IP address in the five-tuples as ID. We use 20M items. The number of bursts of this dataset is 19551 when we set the window size as 40K items. The duration in which the data was collected is 44.02s.

2) Web Page Dataset: The Web Page dataset is built from a collection of web pages, which is downloaded from a website [60]. Each item is 4 bytes long, representing the number of distinct items in a web page. We use 20M items. The number of bursts of this dataset is 6861 when we set the window size as 70K items.

3) Network Dataset: The Network dataset contains users' posting history on the stack exchange website [61]. Each item has three values u, v, t , which means user u answered user v's question at time t. We use u as ID. We use 3M items. The number of bursts of this dataset is 989 when we set the window size as 70K items.

Implementation: The basic version and the optimized version of BurstSketch and the strawman solution are implemented in C++. We run the programs on a server with dual 6-core CPUs (12 threads, Intel Xeon CPU E5-2620 @2.00 GHz) and 64GB DRAM memory. In all experiments, we use MurmurHash3 [62] to implement the hash functions. All related codes of BurstSketch are open-sourced and available at GitHub [63].

Metrics:

1) Recall Rate (RR): The ratio of the number of correctly reported to the number of true instances.

2) Precision Rate (PR): The ratio of the number of correctly reported to the number of reported instances.

3) F1 Score: $\frac{2 \cdot RR \cdot PR}{RR+PR}$. It is calculated from the precision and recall of the test, and it is also a measure of a test's accuracy.

4) Throughput: Million insertions per second (MIPS). We repeat the experiments 5 times and average the results.

B. Experiments on Parameter Settings of BurstSketch

In this subsection, we measure the effects of some key parameters of BurstSketch, namely, the number of hash functions d, the ratio of the memory usage of Stage 1 to the total memory usage $stage\ ratio$, the number of cells in a bucket s, the ratio of the Running Track threshold to the burst threshold l, and the ratio between two adjoin windows for sudden increase or sudden decrease detection k in Stage 2. We also adjust the replacement strategy in Stage 1 and evaluate its effect on the performance of BurstSketch. In the following experiments except the replacement strategy, we set memory to 20 KB, 40 KB and 80 KB, We conduct experiments on the CAIDA dataset, and use F1 score to evaluate the effects of parameters.

Effects of d (Figure 3(a)): *The experiment results show that the best value for* d *is from* 1 *to* 3. In this experiments, we vary the number of hash functions d from 1 to 5. For 20 KB, F1 score peaks when $d = 1$; for 40 KB, F1 score peaks when $d = 2$; for 80 KB, F1 score peaks when $d = 3$. The experimental results that when memory grows larger, more hash functions can achieve better F1 score. However, more hash functions mean more hash computation when inserting, and thus lead to lower throughput. We also find that when d is set to 1, BurstSketch can reach good F1 score, higher than 0.97 when memory is 40 KB and higher than 0.98 when memory is 80 KB. Therefore, we set d to 1 in default.

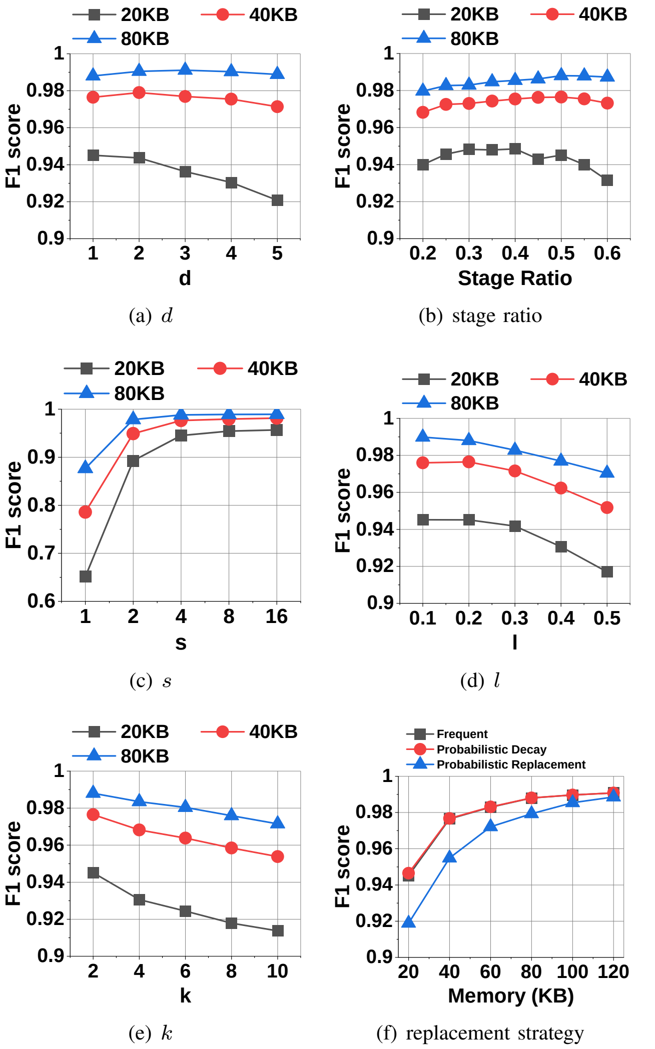
Effects of $stage\ ratio$ (Figure 3(b)): *The experimental results show that the best value for* $stage\ ratio$ *is from* 0.4 *to* 0.5. In this experiment, we vary $stage\ ratio$ from 0.2 to 0.6 in a step of 0.05. The results show that, for 20 KB, F1 score peaks when $stage\ ratio$ = 0.4. For 40 KB and 80 KB, F1 score peaks when $stage\ ratio$ = 0.5. Therefore, the optimal value of $stage\ ratio$ is from 0.4 to 0.5, and we choose 0.5 in default for other experiments.

Effects of s (Figure 3(c)): *The experimental results show that BurstSketch achieves high accuracy when the number of cells in a bucket is* 4. We compare the effects of different values of s. As shown in figure, in all 3 different memory settings, F1 score increases fast as s grows from 1 to 4, and then increases slowly when s grows from 4 to 16. Therefore, when s is set to 4, BurstSketch achieves high F1 score, while larger s does not bring much benefit. So we set s to 4 in default for other experiments.

Effects of l (Figure 3(d)): *The experimental results show that the optimal value for* l *is from* 0.1 *to* 0.2. In this experiment, we compare the performance of BurstSketch when l varies from 0.1 to 0.5. When the memory size is 20 KB and 80 KB, F1 score peaks when $l = 0.1$. When the memory size is 40 KB, F1 score peaks when $l = 0.2$. Thus, the optimal value of l is from 0.1 to 0.2, and we set l = 0.2.

Effects of the ratio k (Figure 3(e)): *Our experimental results show that BurstSketch performs well even when the ratio* k *is very high.* As the ratio k grows from 2 to 10, the F1 score of BurstSketch decreases. However, the drops range between 0.016 and 0.032, which indicates the performance of BurstSketch is stable. For simplicity, we set k to 2 in default in other experiments.

Effects of replacement strategy (Figure 3(f)): *Our experimental results show that the F1 Score of BurstSketch under the three replacement strategies are close.* In this experiment, we compare the effects of three replacement strategies: Frequent, probabilistic decay, and probabilistic replacement. Among



(a) d

(b) stage ratio

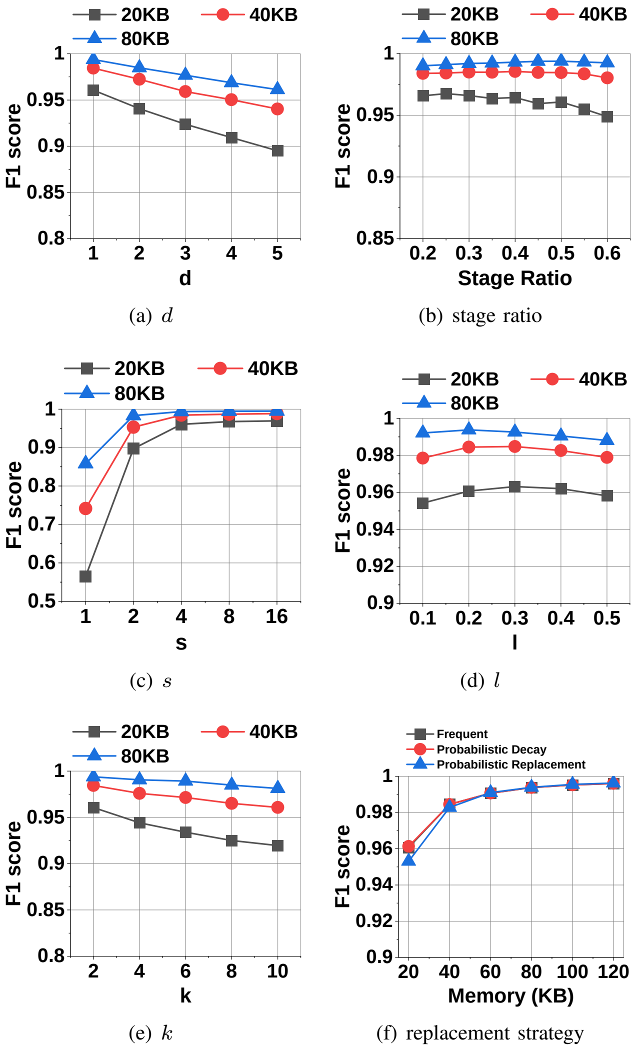
(c) s

(d) l

(e) k

(f) replacement strategy

Fig. 3: Evaluation on parameter settings of BurstSketch.



(a) d

(b) stage ratio

(c) s

(d) l

(e) k

(f) replacement strategy

Fig. 4: Evaluation on parameter settings of the optimization.

three replacement strategies, F1 scores of Frequent and probabilistic decay are sightly higher. Probabilistic replacement is slow and complex, while Frequent is fast and easy to implement. Therefore, we choose Frequent as the replacement strategy for BurstSketch in this paper.

Analysis: Small d can achieve good accuracy, because higher d results in more copies of items, which is memory consuming. For $stage\ ratio$, as memory usage of Stage 1 becomes larger, hash collisions are reduced. If memory usage in Stage 2 is larger, more potential burst items can be monitored at the same time. Therefore, the optimal ratio balances two stages. For l, if it is smaller, items in Stage 1 is easier to be inserted into Stage 2, so that the arrival rate of the item will be more accurate. However, as l becomes smaller, the number of items monitored in Stage 2 grows larger. making Stage 2 easier to be full. Therefore, the optimal ratio balances these two situations.

Concrete Steps for Choosing Parameters: For parameter d, we find that $d = 1$ is a good choice. For parameter $stage\ ratio$, the optimal value is always large than 0.1 in general. Therefore, we can try increasing $stage\ ratio$ to find the optimal value. For parameter s, we can try setting s from 2 to 16 to find the suitable value. For parameter l, the optimal l is always in the range between 0.1 and 0.5 in general. Therefore, we can try setting l from 0.1 to 0.5 to find the optimal value.

C. *Experiments on Parameter Settings of Optimization*

In this subsection, we measure the effects of key parameters for the optimized BurstSketch. The varied parameters in this subsection is the same as the Section VI-B, including d, $stage\ ratio$, s, l, k, and the replacement strategy. In the following experiments except the replacement strategy, we set memory to 20 KB, 40 KB and 80 KB, We conduct experiments on the CAIDA dataset, and use F1 score to evaluate the effects of parameters.

Effects of d (Figure 4(a)): *The experiment results show that the best value for d is 1 for the optimized BurstSketch.* We find that F1 score decreases as d grows for the optimized BurstSketch. Therefore, we set d to be 1 in default for our experiments.

Effects of $stage\ ratio$ (Figure 4(b)): *The experimental results show that the optimal value for the optimized BurstSketch is between 0.25 to 0.5.* For 20 KB, F1 score peaks when $stage\ ratio = 0.25$; for 40 KB, F1 score peaks when $stage\ ratio = 0.4$; for 80 KB, F1 score peaks when $stage\ ratio = 0.5$. Therefore, the optimal value of $stage\ ratio$ is from 0.25 to 0.5, and we choose 0.5 in default for other experiments.

Effects of s (Figure 4(c)): *The experimental results show that the optimized BurstSketch achieves high accuracy when the number of cells in a bucket is* 4. We find that F1 score increases as s grows. When s is set to 4, the optimization achieves high F1 score, while larger s does not bring much benefit. So we set s to 4 in default in our experiments.

Effects of l (Figure 3(d)): *The experimental results show that the optimal value for l is from* 0.2 *to* 0.3. For 20 KB and 40 KB, F1 score peaks when $l = 0.3$; for 80 KB, F1 score peaks when $l = 0.2$. Therefore, the optimal value of l is between 0.2 and 0.3, and we set l to 0.2 in default.

Effects of the ratio k (Figure 3(e)): *Our experimental results show that, the optimized BurstSketch achieves good performance even when k is large.* As the ratio k grows from 2 to 10, the F1 score of BurstSketch drops slightly. For simplicity, we set k to 2 in default in other experiments.

Effects of replacement strategy (Figure 3(f)): *Our experimental results show that the F1 score of the optimization under the three replacement strategies are close.* Considering that Frequent is fast and easy to implement, we choose Frequent as the default replacement strategy.

D. Evaluation of BurstSketch and the optimization.

In this section, we evaluate our BurstSketch and the optimization by comparing them with the strawman solution and prior works. We evaluate the precision, recall, F1 score and throughput on three different datasets: CAIDA, Web Page and Network. For prior works, we choose TopicSketch, CM-PBE-1 and CM-PBE-2. Note that no prior works focusing on the exact same definition of bursts, and we choose above algorithms because they are approximate algorithms, and the definition is similar. The results are shown in Figure 5, 6, 7.

Precision: The experimental results show that, our BurstSketch and the optimized version achieve high precision stably. For three datasets, the precision of BurstSketch and the optimization are close, and both are higher than 93%.

For precision, BurstSketch outperforms the strawman solution and three prior works in all three datasets. In the CAIDA datasets, BurstSketch achieves on average 1.68, 2.01, 7.23 and 7.23 times higher precision than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively. In the Web Page datasets, BurstSketch achieves on average 1.97, 1.60, 8.45 and 8.45 times higher precision than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively. In the Network datasets, BurstSketch achieves on average 1.22, 1.51, 12.51 and 12.51 times higher precision than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively.

Recall: The experimental results show that, our BurstSketch and the optimized version achieve high recall stably. For three datasets, when the memory usage ranges between 20 KB to 120 KB, recall of both algorithms is higher than 87%. Compared with the basic version, the optimized BurstSketch can achieve higher recall. On average, the optimized BurstSketch achieves 1.016x recall in CAIDA, 1.027x recall in Web Page, 1.015x recall in Network.

For recall, BurstSketch also outperforms the strawman solution and three prior works in all three datasets. In the CAIDA datasets, BurstSketch achieves on average 3.97, 1.60, 1.24 and 1.24 times higher recall than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively. In the Web Page datasets, BurstSketch achieves on average 20.70, 9.49, 1.38 and 1.38 times higher recall than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively. In the Network datasets, BurstSketch achieves on average 2.75, 1.04, 1.26 and 1.26 times higher recall than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively.

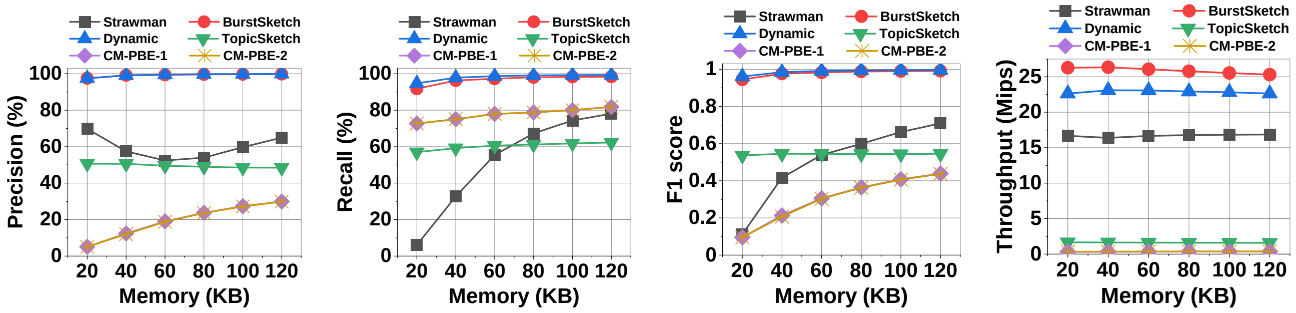
F1 score: The experimental results show that, our BurstSketch and the optimized version achieve high F1 score stably. For three datasets, when the memory usage ranges between 20 KB to 120 KB, the F1 score of both algorithms is higher than 0.90. Compared with the basic version, the optimized BurstSketch can achieve higher F1 score. On average, the optimized BurstSketch achieves 1.008x F1 score in CAIDA, 1.017x recall in Web Page, 1.008x recall in Network.

For F1 score, BurstSketch outperforms the strawman solution and three prior works in all three datasets. In the CAIDA datasets, BurstSketch achieves on average 2.86, 1.80, 4.17 and 4.17 times higher F1 score than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively. In the Web Page datasets, BurstSketch achieves on average 11.63, 5.59, 4.85 and 4.85 times higher F1 score than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively. In the Network datasets, BurstSketch achieves on average 2.00, 1.27, 6.77 and 6.77 times higher F1 score than strawman solution, TopicSketch, CM-PBE-1 and CM-PBE-2, respectively.

Throughput: The experimental results show that, our BurstSketch and the optimized version achieve high throughput. For three datasets, when the memory usage ranges between 20 KB to 120 KB, the throughput of the basic BurstSketch ranges between 25.12 and 27.16 Mips, and the throughput of the optimized BurstSketch ranges between 22.49 and 24.89 Mips. Compared with the basic version, the optimized BurstSketch has slower throughput, because the optimization sacrifices insertion speed for higher accuracy.

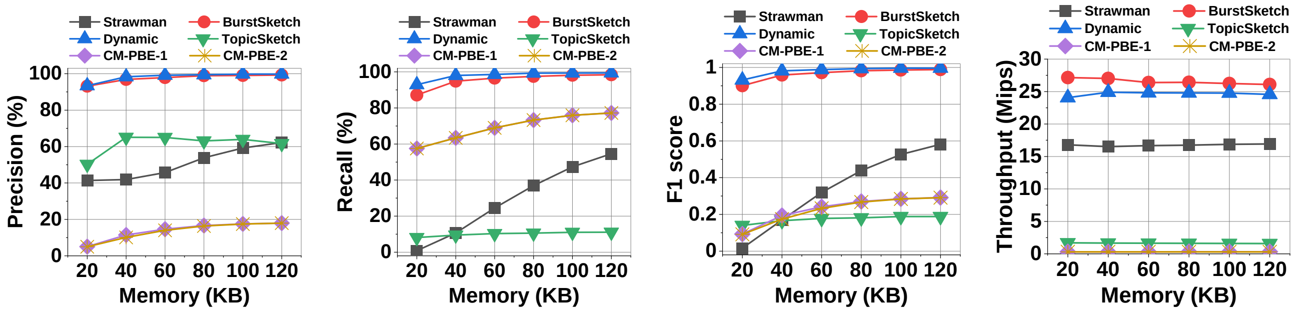
For all algorithms, the throughput is similar in all three datasets. BurstSketch achieves much faster speed than the strawman solution and three prior works. BurstSketch achieves 1.56x higher throughput than strawman solution on average, 16.14x higher throughput than TopicSketch on average, and 79.98x higher throughput than CM-PBE-1 and CM-PBE-2 on average.

Summary: BurstSketch achieves high F1 score and throughput, and the performance is stable in all three datasets. The F1 score and throughput of BurstSketch is much higher than the strawman solution and three prior works, which indicates that BurstSketch is much more accurate and faster. The optimized version sacrifices the processing speed for higher recall rate and F1 score. The experimental results show that its throughput is slightly slower but still higher than 20 MIPS, and it improves the recall rate by up to 6%. It is noticeable that the recall rate of the basic version is already high, and therefore the improvement of the optimization is nontrivial.



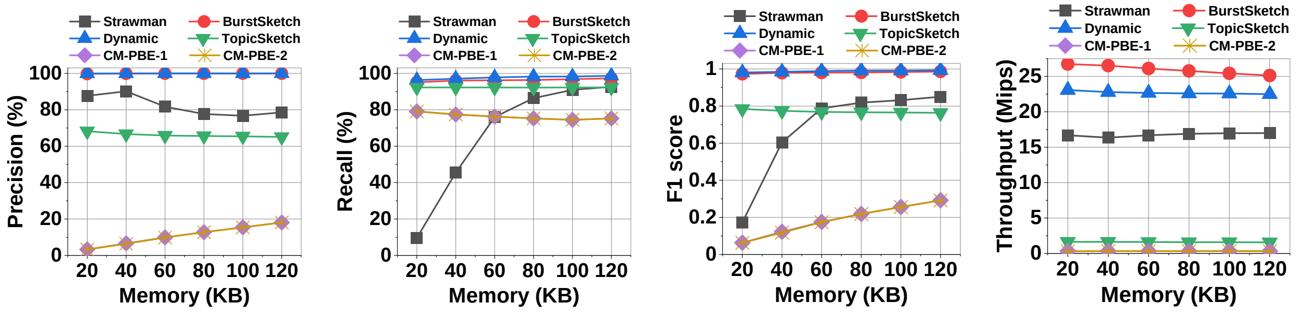
(a) Precision (b) Recall (c) F1 score (d) Throughput

Fig. 5: Experiments on CAIDA datasets.



(a) Precision (b) Recall (c) F1 score (d) Throughput

Fig. 6: Experiments on Web Page datasets.



(a) Precision (b) Recall (c) F1 score (d) Throughput

Fig. 7: Experiments on Network datasets.

E. Analysis on BurstSketch and Optimization

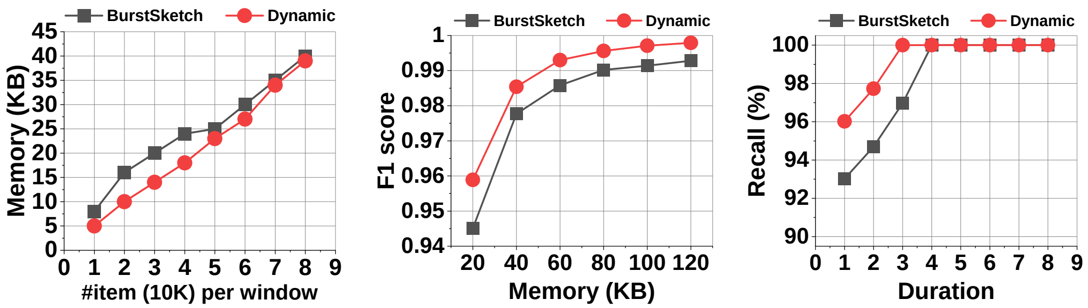
In this section, we analyse BurstSketch and the optimization from several aspects. First, we evaluate the minimal memory usage to achieve an acceptable performance in data streams of different speeds. Second, we evaluate the performance in detecting bursts inside bursts. Third, we evaluate how the duration of the burst effects the performance. Fourth, we compare their performances in time-based windows and count-based windows. Finally, we measure the effectiveness of Stage 1 in BurstSketch and the optimization.

Memory usage in burst detection in data streams of different speed (Figure 8(a)): In this experiment, we vary the speed of the input data stream (from 10K items to 80K items per window), and check how much memory BurstSketch and the optimization have to use to achieve an F1 score of 0.95. The experimental results show that the memory usage to achieve an F1 score of 0.95 grows linearly with the increase of the speed of the data stream. And optimized BurstSketch require less memory.

Bursts inside bursts (Figure 8(b)): The results show that BurstSketch and the optimization performs well in detecting bursts inside bursts. With 20 KB, BurstSketch and the optimization can achieve an F1 score higher than 0.94. And the F1 score grows rapidly as memory increases for both BurstSketch and the optimization.

The influence of the duration of bursts (Figure 8(c)): The experimental results reveal that as the duration of burst grows larger, the RR of BurstSketch and the optimization increases. The reason is that the streams with larger duration tend to be stable, and our algorithm detects this kind of bursts more effectively.

Performance under time-based and count-based windows (Figure 9(a)): Different from count-based windows, the number of items per window could vary a lot in time-based windows. The experimental results show that the performance under count-based windows is slightly lower than that under

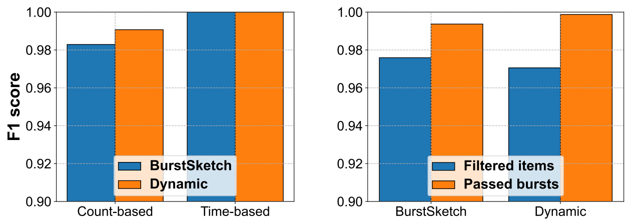


(a) memory usage

(b) bursts inside bursts

(c) bursts' duration

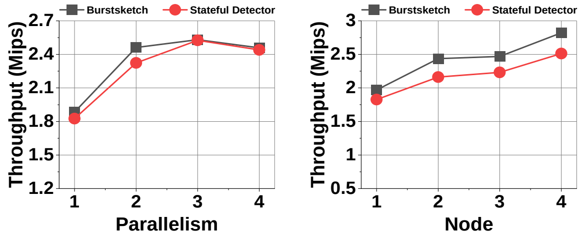
Fig. 8: Analyses on BurstSketch and the optimization, including memory usage for different data stream speed, detecting burst inside burst, and performance under different bursts' duration.



(a) window type

(b) effectiveness of Stage 1

Fig. 9: Analyses on BurstSketch and the optimization, including window type and the effectiveness of Stage 1.



(a) Local

(b) Cluster

Fig. 10: Comparison between BurstSketch and Stateful Detector on Apache Flink

time-based windows. This reveals that the accuracy of our BurstSketch and the optimization is insensitive to whether the number of items in each window is equal. The reason behind is that, no matter whether the number of items in each window is equal, after the items are filtered by Stage 1, the number of items (potential bursts) that reach Stage 2 varies a lot per window.

Effectiveness of Stage 1 (Figure 9(b)): In this experiments we measure the ratio of filtered items to the whole items (Filtered items), and the ratio of bursts that pass Stage 1 to all bursts (Passed bursts). The experimental results show that Stage 1 is highly effective in filtering out non-burst items, since more than 97% of the items in the data stream are filtered out. More than 99% real bursts can pass Stage 1, which shows that Stage 1 has a very high recall rate. Moreover, we find that with optimization, BurstSketch filters less items but let more bursts go through, which explains why the optimization achieves better performance.

F. *Experiments on Apache Flink*

In this section, we compare our BurstSketch with Stateful Detector on Apache Flink.

Experimental setup: We conduct cluster experiments and local experiments, separately. For cluster experiments, our cluster consists of one master node and 4 worker nodes. Each node has 4 virtual CPU cores of Intel XEON Platinum 8369B, and 8 GB main memory. For local experiments, we only conduct them in the local mode of master node. For both cluster and local experiments, we configure the memory size of job manager and task manager to be 1 GB, and run 5 times to compute average throughput for BurstSketch and Stateful Detector. The provided data is constructed based on CAIDA [59] datasets. we deploy a Hadoop Distributed File System (HDFS) in our Flink cluster as the data source, in which we set the master node as NameNode and the worker nodes as DataNodes. In Flink experiments, each node uses Flink 1.13.1, Java 11 and Hadoop 2.8.3 running on Ubuntu 20.04 LTS.

Local Experiments (Figure 10(a)): We vary the parallelism from 1 to 4 and evaluate the throughput. The experimental results show that BurstSketch achieves $1.03x$ throughput higher than Stateful Detector. When the parallelism grows from 1 to 3, the throughput of BurstSketch increases from 1.885 Mips to 2.531 Mips, and that of Stateful Detector increases from 1.825 Mips to 2.526 Mips. When the parallelism grows from 3 to 4, the throughput of BurstSketch drops from 2.531 Mips to 2.459 Mips, and that of Stateful Detector drops from 2.526 Mips to 2.440 Mips. The reason why throughput drops may be that the master node has only 4 CPU cores, and there are other process such as job manager, Hadoop NameNode running in the node. When the parallelism is 3, the CPU utilization is already high, and the increase of parallelism can only lead to the drop of throughput.

Cluster Experiments (Figure 10(b)): We vary the number of worker nodes in the Flink cluster from 1 to 4 and evaluate the throughput. As shown in figure, when the number of worker nodes increases from 1 to 4, the throughput of BurstSketch increases from 1.969 Mips to 2.821 Mips, and that of Stateful Detector increases from 1.826 Mips to 2.513 Mips. We find that in cluster experiments, the throughput of BurstSketch is $1.11x$ higher than that of Stateful Detector.

Analysis: The experimental results show that the throughput of BurstSketch increases when the parallelism and the number of worker nodes grow in reasonable ranges. Besides, BurstSketch achieves higher throughput than Stateful Detector in both local and cluster settings. The reason behind is that, Stateful Detector should maintain a state for each key, and access the corresponding state each time an item is processed, and the size of all states can be large when the dataset is large. BurstSketch, however, only maintains and accesses limited memory for each parallel instance. Therefore, compared with Stateful Detector, BurstSketch better utilizes the caches, improving the performance of throughput.

VII. Conclusion

Real-time burst detection in high-speed data streams is important in many applications. This paper proposes a novel algorithm called BurstSketch for real-time burst detection, which is fast, memory-efficient, and accurate. Experimental results show that BurstSketch can achieve high accuracy with fairly limited memory usage in real-time burst detection for high-speed items.

Acknowledgment

This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179,61832001).

References

[1] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, “Burstsketch: Finding bursts in data streams,” in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2375–2383.

[2] J. Kleinberg, “Bursty and hierarchical structure in streams,” *KDD*, 2003.

[3] Q. He, K. Chang, and E.-P. Lim, “Using burstiness to improve clustering of topics in news streams,” in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 2007, pp. 493–498.

[4] Q. He, K. Chang, E.-P. Lim, and J. Zhang, “Bursty feature representation for clustering text streams,” in *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM, 2007, pp. 491–496.

[5] M. Vlachos, C. Meek, Z. Vagena, and D. Gunopulos, “Identifying similarities, periodicities and bursts for online search queries,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 131–142.

[6] T. Lappas, B. Arai, M. Platakis, D. Kotsakos, and D. Gunopulos, “On burstiness-aware search for document sequences,” in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 477–486.

[7] G. Dong, W. Yang, F. Zhu, and W. Wang, “Discovering burst patterns of burst topic in twitter,” *Computers & Electrical Engineering*, vol. 58, pp. 551–559, 2017.

[8] D. Paul, Y. Peng, and F. Li, “Bursty event detection throughout histories,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1370–1381.

[9] W. Xie, F. Zhu, J. Jiang, E.-P. Lim, and K. Wang, “Topicsketch: Real-time bursty topic detection from twitter,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 8, pp. 2216–2229, 2016.

[10] N. Parikh and N. Sundaresan, “Scalable and near real-time burst detection from ecommerce queries,” in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 972–980.

[11] L. Golab, D. DeHaan, E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro, “Identifying frequent items in sliding windows over on-line packet streams,” in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003, pp. 173–178.

[12] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, “Heavyguardian: Separate and guard hot items in data streams,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2584–2593.

[13] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, “Randomized admission policy for efficient top-k and frequency estimation,” in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[14] P. Roy, A. Khan, and G. Alonso, “Augmented sketch: Faster and more accurate stream processing,” in *SIGMOD*, 2016.

[15] X. Zhang and D. Shasha, “Better burst detection,” in *22nd International Conference on Data Engineering (ICDE’06)*. IEEE, 2006, pp. 146–146.

[16] Y. Zhu and D. Shasha, “Efficient elastic burst detection in data streams,” in *SIGKDD*, 2003.

[17] Z. Yuan, Y. Jia, and S. Yang, “Online burst detection over high speed short text streams,” in *ICCS*, 2007.

[18] R. Maison and M. Zakrzewicz, “Prediction-based load shedding for burst data streams,” *Bell Labs Technical Journal*, 2011.

[19] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin, “Sliding-window top-k queries on uncertain streams,” *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 301–312, 2008.

[20] Y. Tong, L. Chen, Y. Cheng, and P. S. Yu, “Mining frequent itemsets over uncertain databases,” *arXiv preprint arXiv:1208.0292*, 2012.

[21] Y. Tong, X. Zhang, and L. Chen, “Tracking frequent items over distributed probabilistic data,” *World Wide Web*, vol. 19, no. 4, pp. 579–604, 2016.

[22] Y. Tong, L. Chen, and P. S. Yu, “Ufimt: an uncertain frequent itemset mining toolbox,” in *Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining*, 2012, pp. 1508–1511.

[23] Y.-X. Tong, L. Chen, and J. She, “Mining frequent itemsets in correlated uncertain databases,” *Journal of Computer Science and Technology*, vol. 30, no. 4, pp. 696–712, 2015.

[24] Y. Tong, L. Chen, and B. Ding, “Discovering threshold-based frequent closed itemsets over probabilistic data,” in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 270–281.

[25] G. Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications,” *Journal of Algorithms*, 2005.

[26] C. Estan and G. Varghese, “New directions in traffic measurement and accounting,” *ACM SIGMCOMM CCR*, 2002.

[27] M. Charikar, K. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” in *Automata, Languages and Programming*, 2002.

[28] A. Shrivastava, A. C. Konig, and M. Bilenko, “Time adaptive sketches (ada-sketches) for summarizing data streams,” in *SIGMOD*, 2016.

[29] Q. Huang, S. Sheng, X. Chen, Y. Bao, R. Zhang, Y. Xu, and G. Zhang, “Toward {Nearly-Zero-Error} sketching via compressive sensing,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 1027–1044.

[30] H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui, “Stingy sketch: a sketch framework for accurate and fast frequency estimation,” *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1426–1438, 2022.

[31] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, “Elastic sketch: adaptive and fast network-wide measurements,” in *SIGCOMM*, 2018.

[32] A. Metwally, D. Agrawal, and A. El Abbadi, “Efficient computation of frequent and top-k elements in data streams,” in *ICDT*, 2005.

[33] D. Ting, “Data sketches for disaggregated subset sum and frequent item estimation,” in *SIGMOD*, 2018.

[34] G. Cormode and S. Muthukrishnan, “What’s new: Finding significant differences in network data streams,” *IEEE/ACM Transactions on Networking*, 2005.

[35] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik, “Reversible sketches: enabling monitoring and analysis over high-speed data streams,” *IEEE/ACM Transactions on Networking*, 2007.

[36] K. Balachander, S. Subhabrata, Z. Yin, and C. Yan, “Sketch-based change detection: methods, evaluation, and applications,” in *SIGCOMM*, 2003.

[37] N. Tang, Q. Chen, and P. Mitra, “Graph stream summarization: From big bang to big crunch,” in *SIGMOD*, 2016.

[38] X. Gou, L. Zou, C. Zhao, and T. Yang, “Graph stream sketch: Summarizing graph streams with high speed and accuracy,” *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[39] K. S. Tai, V. Sharan, P. Bailis, and G. Valiant, “Sketching linear classifiers over data streams,” in *SIGMOD*, 2018.

[40] G. Cormode, “Sketch techniques for approximate query processing,” *TRDB*, 2011.

[41] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. Lui, and X. Guan, “A memory-efficient sketch method for estimating high similarities in streaming sets,” in *SIGKDD*, 2019.

[42] Y. Wang, P. Lin, and Y. Hong, “Distributed regression estimation with incomplete data in multi-agent networks,” *Science China Information Sciences*, 2018.

[43] T. Zheng, G. Chen, and et al., “Real-time intelligent big data processing: technology, platform, and applications,” *Science China Information Sciences*, 2019.

[44] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, “Finding persistent items in data streams,” *VLDB Endowment*, 2016.

[45] D. Ting, “Count-min: Optimal estimation and tight error bounds using empirical error distributions,” in *SIGKDD*, 2018.

[46] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, “A general-purpose counting filter: Making every bit count,” in *Proceedings of the 2017 ACM international conference on Management of Data*, 2017, pp. 775–787.

[47] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. Lui, and X. Guan, “A memory-efficient sketch method for estimating high similarities in streaming sets,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 25–33.

[48] Y. Zhao, Y. Zhang, Y. Li, Y. Zhou, C. Chen, T. Yang, and B. Cui, “Minmax sampling: A near-optimal global summary for aggregation in the wide area,” in *Proceedings of the 2022 International Conference on Management of Data*, 2022.

[49] J. Jiang, F. Fu, T. Yang, and B. Cui, “Sketchml: Accelerating distributed machine learning with data sketches,” in *SIGMOD*, 2018.

[50] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, “Persistent bloom filter: Membership testing for the entire history,” in *SIGMOD*, 2018.

[51] Z. Wei, G. Luo, K. Yi, X. Du, and J.-R. Wen, “Persistent data sketching,” in *SIGMOD*, 2015.

[52] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[53] R. Poepsel-Lemaitre, M. Kiefer, J. von Hein, J.-A. Quiané-Ruiz, and V. Markl, “In the land of data streams where synopses are missing, one framework to bring them all,” *Proceedings of the VLDB Endowment*, vol. 14, no. 10, pp. 1818–1831, 2021.

[54] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, “Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm,” in *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[55] C. Masson, J. E. Rim, and H. K. Lee, “Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees,” *arXiv preprint arXiv:1908.10693*, 2019.

[56] J. Chakkaradhari, “Large scale centrality measures in apache flink and apache giraph,” Ph.D. dissertation, Technical University of Berlin, 2014.

[57] S. Katragadda, R. Gottumukkala, M. Pusala, V. Raghavan, and J. Wojtkiewicz, “Distributed real time link prediction on graph streams,” in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 2912–2917.

[58] A. Kontaxakis, N. Giatrakos, and A. Deligiannakis, “A synopses data engine for interactive extreme-scale analytics,” in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 2085–2088.

[59] “The caida anonymized 2016 internet traces.” http://www.caida.org/data/overview/.

[60] “Real-life transactional dataset.” http://fimi.ua.ac.be/data/.

[61] “The Network dataset Internet Traces.” http://snap.stanford.edu/data/.

[62] “Murmurhash,” https://github.com/aappleby/smhasher.

[63] “Source code related to BurstSketch,” https://github.com/mrj222/burst-sketch.

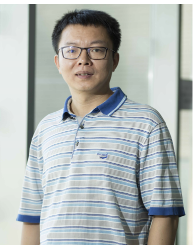
Ruijie Miao
Ruijie Miao is an undergraduate student of Peking University. His major is Computer Science. His research interests include network measurement and streaming algorithms.

Zheng Zhong
Zheng Zhong is an undergraduate student of Peking University. His research interests include Streaming processing, Bloom filters, and data structure. He published papers in SIGMOD and SIGKDD. He is also the reviewer of SIGKDD2022.

Jiarui Guo
Jiarui Guo is an undergraduate student of Peking University majoring in Computer Science. His research interests include approximation algorithms in data streams and computer network systems.

Zikun Li
Zikun Li received B.S. degree in Computer Science from Peking University in 2021. His research interests include networked systems, database and machine learning.

Tong Yang
Tong Yang received the PhD degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with School of Computer Science, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, and KV stores. He has published more than ten papers in SIGCOMM, SIGKDD, SIGMOD, NSDI, etc.



Bin Cui
Bin Cui is a professor in School of Computer Science and Director of Institute of Network Computing and Information Systems, at Peking University. His research interests include database system architectures, query and index techniques, big data management and mining. He is a senior member of IEEE, member of ACM and distinguished member of CCF.